CLEMSON
School of COMPUTING

Amortized analysis

Amortized analysis

- In an amortized analysis, we estimate the time required to perform a sequence of operations.

- With amortized analysis, we have the opportunity to show that the average cost of an operation is small, even if the cost of such an operation looks big (according to asymptotic notation)

Amortized analysis

Two techniques

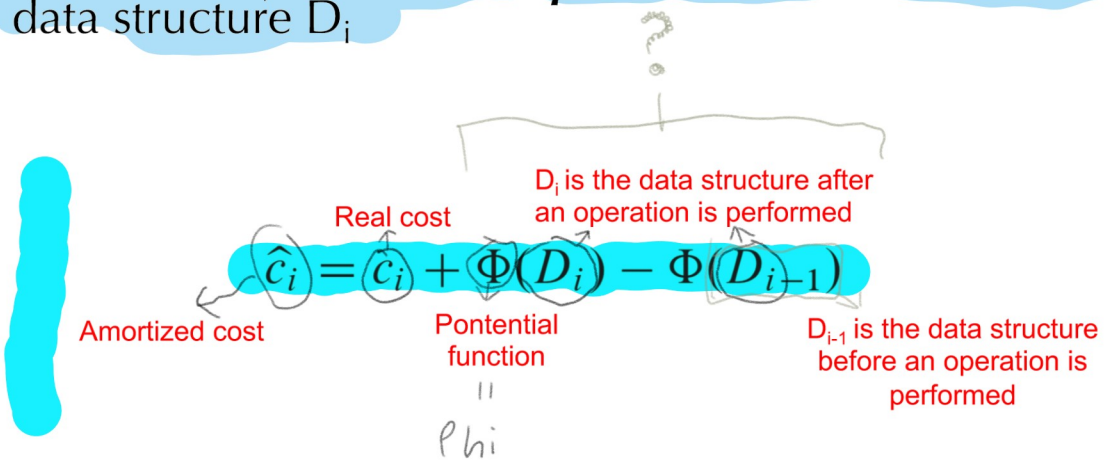
- Accounting method
  - Faster to use, intuitive. Better suited for simple operations

- **Potential method**
  - **More formal and structured. Better suited for involved algorithms**

The potential method

- Instead of representing prepaid work as credit stored with specific objects in the data structure, the ***potential method*** of amortized analysis represents the prepaid work as "potential energy," or just "potential," which can be released to pay for future operations.

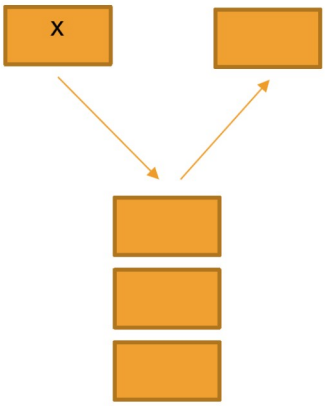
- A ***potential function*** maps each data structure D_i to a real number, which is the ***potential*** associated with the data structure D_i



ci hat = credits earned from algorithm
ci = real cost of algorithm (like, RAM model or whatever)
Di = data structure *after* algorithm completed on it
chi(Di) = "potential function", a function that scales with algorithm complexity using some element(s) of data structure (Di) as *input* (not necessarily multiplication)
D (i - 1) = supposedly the data structure before the algorithm is used on it but I don't know why the equation is -1 and not a variable (nvm its because its the step before i but i dont know what i is) (edited)

First example

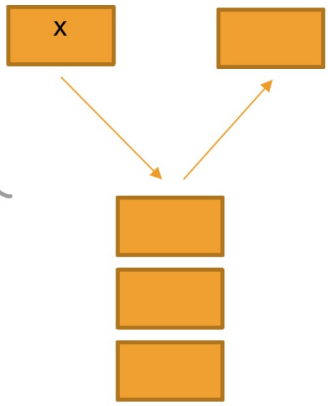
Consider a stack data structure



Step 1 – find the potential function more suitable for our problem.

Idea – the potential function should return a high number when we are close to perform an expensive operation

First example - Stack



ϕ = returns the number of element in the stack

- Thats what determines expensive operations

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Cost of function

Elements after push

Elements before push

PUSH - = 1 + (k'+1) - k'
= 2

Elements after pop

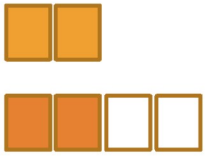
Elements before pop

POP = 1 + (k'-1) – k'
= 0

MULTIPOP = k + (k' - k) - k'
= 0

Second example – Dynamic Table

$\phi = 2*T.num - T.size$



0 when T is half empty

T.Num when T is full

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

INSERT - = 1 + (2*(T.num + 1) - T.size) - (2*T.num – T.size)
= 1 + 2 = 3

RESIZE = (T.num+1) + (2*(T.num+1) - 2*T.size) – (2*T.num – T.size)
= T.size +1 + 2T.size + 2 - 2T.size – 2T.size + T.size
= 3

What's next?

- Book chapters
  - Chapter 1.4
- In class activities
  - Use amortized analysis on new examples
- Next class
  - Union find data structures

Amortized Analysis

Exercise 1

- Consider a ***clearable table*** data structure that supports the following two methods:
  - $add(e)$: add the element e to the next available cell in the table
  - $clear()$: empty the table by removing all its elements

- Let S be a clearable table with n elements, what is the cost of the $clear()$ operation?
- Suppose a series of n operations is performed on an initially empty clearable table, what is the total cost?

Exercise 1

- Consider a ***clearable table*** data structure that supports the following two methods:
  - $add(e)$: add the element e to the next available cell in the table
  - $clear()$: empty the table by removing all its elements

- Let S be a clearable table with n elements, what is the cost of the $clear()$ operation? $O(n)$
- Suppose a series of n operations is performed on an initially empty clearable table, what is the total cost? $O(n^2) \rightarrow O(n)$

Exercise 1

- Use accounting method to show that the ***amortized running time*** of each operation of a clearable table structure is $O(1)$.
  - $add(e)$: add the element e to the next available cell in the table
  - $clear()$: empty the table by removing all its elements

Exercise 1

- Use accounting method to show that the ***amortized running time*** of each operation of a clearable table structure is $O(1)$.
  - $add(e)$: add the element e to the next available cell in the table
  - $clear()$: empty the table by removing all its elements

	Real cost	Schema	Amortized cost
$add(e)$	1	2	O(1)
$clear()$	n	0	O(1)

- For each $add()$ operation, we charge 2 coins and leave 1 credit for the $clear()$ operation.

Exercise 1

- Use potential function to show that the ***amortized running time*** of each operation of a clearable table structure is $O(1)$.
  - $add(e)$: add the element e to the next available cell in the table
  - $clear()$: empty the table by removing all its elements

Exercise 1

- Use the potential function to show that the ***amortized running time*** of each operation of a clearable table structure is $O(1)$.
  - Let $\Phi = number\ of\ elements\ in\ the\ table$
  - $add()$ operation: $t_i' = t_i + \Phi_i - \Phi_{i-1} = 1 + (k+1) - k = 2$
  - $clear()$ operation: $t_i' = t_i + \Phi_i - \Phi_{i-1} = k + 0 - k = 0$

Exercise 1

- Show that the total time cost of performing a series of n operations on an initially empty clearable table is $O(n)$.
  - $T = \sum_{i=1}^{n} t_i = \sum_{i=1}^{n} (t_i' + \Phi_{i-1} - \Phi_i)$
  - $= \sum_{i=1}^{n} t_i' + \sum_{i=1}^{n} (\Phi_{i-1} - \Phi_i)$
  - $= T' + \sum_{i=1}^{n} (\Phi_{i-1} - \Phi_i)$
  - $= T' + \Phi_0 - \Phi_n = T' - \Phi_n$ (the number of elements left in the table)
  - Given that $T \leq T'$ and $T' = O(n)$, the total time cost is $O(n)$.

Exercise 2

- ***Queue with two stacks.*** Implement a queue with two stacks so that each queue operation (i.e., *enqueue* and *dequeue* operation) takes a **constant amortized** number of stack operations.

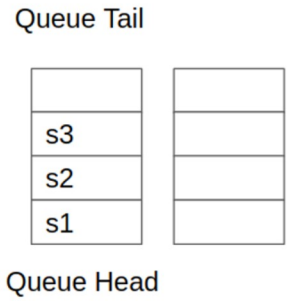
Exercise 2

- ***Queue with two stacks.*** Implement a queue with two stacks so that each queue operation (i.e., *enqueue* and *dequeue* operation) takes a **constant amortized** number of stack operations.

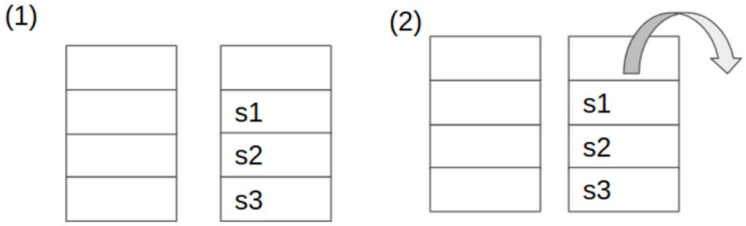
- *Hint*: If you push elements onto a stack and then pop them all, they appear in reverse order. If you repeat this process, they will be back in order.

Efficient Implementation

EnQueue s1, s2, s3



Dequeue() (should return s1)



Exercise 2

Queue with two stacks.

```
enqueue(e):
  S_in.push(e)
```

```
dequeue():
  if(S_out.empty()):
    while(!S_in.empty()):
      S_out.push(S_in.pop())
  return S_out.pop()
```

Use amortized analysis to prove the amortized cost of each operation is $O(1)$.